# A CORBA Extension for Intelligent Software Environments

Robert E. Filman[††], David J. Korsmeyer*[†], Diana D. Lee[††]

{ rfilman | dkorsmeyer | ddlee }@mail.arc.nasa.gov

[†]NASA Ames Research Center, MS/269-3

[††]NASA Ames Research Center, Caelum Research Corp., MS/269-1

Moffett Field, CA 94305

## Abstract

*We describe the implementation of a technology that achieves system-wide properties in large software systems by controlling and modifying inter-component communications. Traditional component-based applications intermix the code for component functionality with support for systematic properties. This produces non-reusable components and inflexible systems. The Object Infrastructure Framework (OIF) separates systematic properties from functional code and provides a mechanism for weaving them together with functional components. This allows a much richer variety of component reuse and system evolution. Key elements of this technology include intercepting inter-component communications with discrete, dynamically configurable "injectors," annotating communications and processes with additional meta-information, and a high-level, declarative specification language for describing the mapping between desired system properties and services that achieve these properties. We have implemented these ideas in a CORBA/Java framework for distributed computing, and are currently applying them to a distributed system for the analysis of aerospace design (wind-tunnel, and CFD) data.*

## Keywords

Object Infrastructure Framework, distributed computing, middleware, frameworks, CORBA, injectors, Intelligent Synthesis Environment, ility, security, reliability, manageability, quality of service

## Intelligent Engineering Software Environments

> "The ISE [Intelligent Synthesis Environment] aims to link scientists, design teams, manufacturers, suppliers, and consultants in the creation and operation of an aerospace system and in synthesizing its missions. The ultimate goal is to significantly increase creativity and knowledge and eventually dissolve rigid cultural boundaries among diverse engineering and science teams."
> — Goldin, Venneri and Noor [1]

NASA programs and missions are widely dispersed among NASA centers and contractors. Typically this has implied a geographic centralization of skills required for a mission to be completed. As we enter the 21st century, the need for more productive engineering environments will greatly change the way we engineer systems. No longer do we have the option of slow design cycles and separate component engineering for aerospace systems. The mantra "faster, cheaper, better" is required by the funding realities, the high quality, and rapid mission requirements of today programs. As Goldin, *et. al* have observed, we need Intelligent Synthesis Environments that allow engineering tools to be freely applied to the *virtual* design and analysis of NASA products.

*Virtual* design implies the capability to assess and analyze the impact and variation of design decisions without developing (or with very limited development of) hardware prototypes. This distributed and collaborative design simulation is within the technical range of existing design tools except for the fact that these tools tend to be hand-crafted, finely tuned, and built to operate in a unique and well formulated computational environments. Intelligent synthesis requires something else.

Traditionally, we built engineering analysis and design tools for specific computing architectures and purposes. Any distribution, sharing, or composition of such elements was explicitly part of their design and laboriously constructed to the particular architecture and topology available. Today we are seeing the emergence of "software bus architectures" that allow well-defined components to provide services accessible from anywhere on a network. The most prominent (and simplest) of such systems is the World Wide Web, which serves (in its most straightforward application) to provide fixed and parameterized "documents" in response to requests. While this works well for web-surfing and shopping, more complex applications require invoking a greater variety of actions on complex data. Such applications demand application program interfaces. The state-of-the-art in providing such remote application interfaces is *Distributed Object Technology* (DOT), where the system presents to the programmer the illusion that a remote object or functionality is local to the programmer's environment, and can be called and manipulated like any other local object. Examples of DOT technology include CORBA, DCOM, and Java RMI.

DOT allows distributed components to invoke each other and communicate, but is only a first step towards an Intelligent Synthesis Environment. Users and contributors to a system of shared data and tools will demand appropriate policies on issues such as security, reliability, data ownership, quality of service, and management and monitoring of processes. (We call such qualities *ilities*.) In the following sections, we briefly overview the implementation of DOT, describe a technology (discrete behavior injectors on the communication paths between components) for extending DOT to allow dynamic management of ilities, describe some of our experience in implementing such injectors, and touch on the application of this technology to intelligent synthesis environments.

## Distributed Object Technology

How are distributed applications built? Current technology uses sockets, messages and events, remote procedure calls (e.g., DCE), or Object Request Brokers (ORBs, including, for example, CORBA [2], JavaRMI and DCOM). Without too much loss of generality, we focus on ORB frameworks and use CORBA as our exemplar.

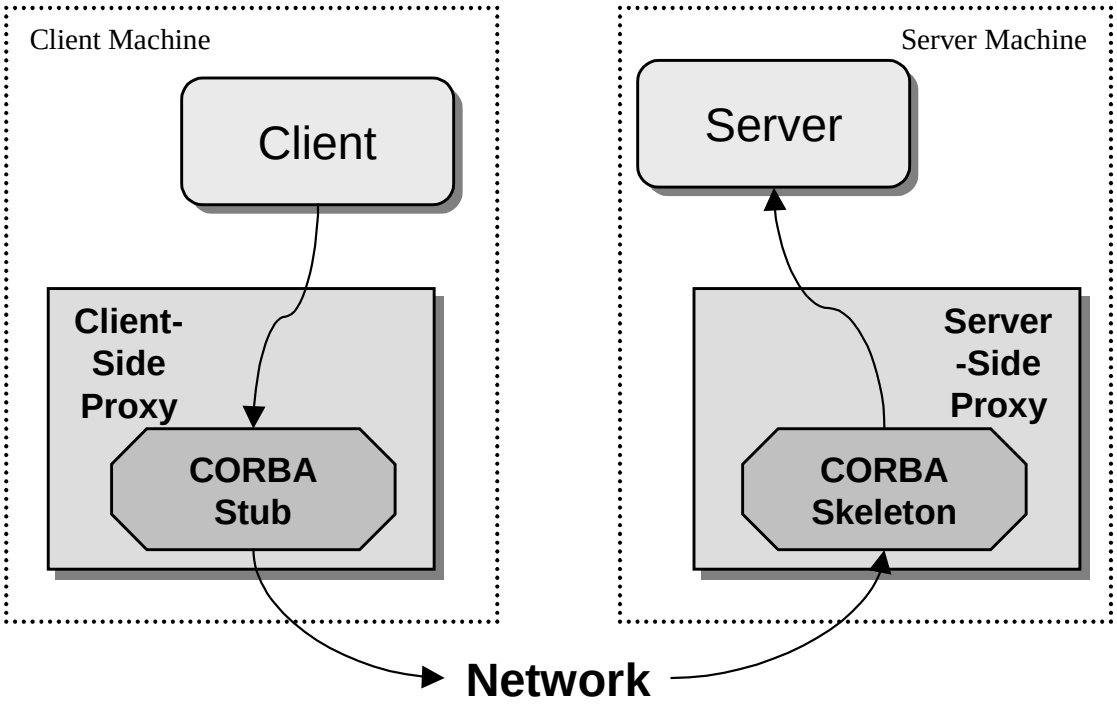


Figure 1: CORBA stubs and skeletons

ORBs allow object-oriented applications to use objects that are not in the same address space. The key semantic advantage of ORBs is that ORBs provide *location transparency*—to an application program, a remote object "looks like" any other (local) object. This is accomplished by delegation to "proxy" objects. On the client side of the application, we create a proxy object for the server (the *stub*). (Here we use "client" and "server" generically as the originator and recipient of a request.) From the point of view of the client code, the stub is an object like any other object that supports the interface of the server. The stub is responsible for accepting calls from client programs, and translating these calls into a "linear form" that can be transmitted over a network (*marshaling*). It sends them over the net to the server machine, where a corresponding proxy (the *skeleton*) re-inflates that linear form into object references and primitive data types (*demarshaling*) and calls the appropriate method on the actual server implementation object. The inverse process is used for the return value. Marshaling handles all issues relating to transmission and translation of the various data types. To the application code, this process is transparent: the client is unaware that the client side proxy is delegating the request to a remote implementation object and the server does not suspect that it is being used by a remote call. Figure 1 illustrates the ORB architecture.

The client and server share the definition of the server interface through an Interface Definition Language (IDL). A CORBA vendor provides compilers that take IDL and generate the code for stub and skeleton proxies in the target languages. (One of the major virtues of CORBA is its multilingual capabilities. A client written in one programming language can, relatively seamlessly, invoke services implemented in another, with data

representations translated automatically. CORBA is thus a favorite of those trying to integrate multi-lingual, distributed, legacy applications into a coherent whole.)

ORB technology provides object location transparency and hides the details of marshaling and communication protocols. What it doesn't do is handle issues of partial failures, security, quality of service and other such ility concerns. ORBs such as CORBA and Enterprise Java Beans provide different discrete mechanisms for particular ility issues, but such mechanisms typically (1) provide only a finite number of choices for the application architect, and (2) require a good understanding and diligent application of the mechanism by the application programmer.

## The Object Infrastructure Framework

The implementation heart of OIF is inserting behavior on the communication path between components. This effectively serves to "wrap" services with additional actions at both the client and server ends. The OIF wrapping mechanism is distinguished by the following features: (1) OIF wrappers are composed of discrete "injectors" that are first class objects. Injectors can be sequenced, combined, treated uniformly by utilities, and dynamically manipulated as a program is running. (2) Wrapping is by object/method. That is, a specific object (or proxy) can have its own unique set of injectors for a particular method. (3) Injectors communicate among themselves and with the client and server by *annotations*, meta information about the call and result. Typical annotations include session identification, request priority, sending and due dates, version and configuration, futures, cyber wallet, public key, sender identification and conversational thread. OIF propagates this meta information into the thread of a called service and from there to the annotations of that service's calls. (4) OIF provides a high-level specification compiler (Pragma) that takes a description of the desired properties of an application and how to achieve these properties and arranges these behaviors on the appropriate wrappers. Figure 2 illustrates the relationship of injectors to CORBA skeletons and stubs.

The motivation of these features and their relationship to the Pragma compiler is discussed in the paper *Inserting Ilities by Controlling Communications* [3]. Here we consider the implementation mechanism in greater detail.

## Injectors

In general, the programmer uses Pragma to define a default sequence of injectors for a given interface and method. In the course of program execution, it is possible (with the appropriate privileges) to modify the injector sequence for any particular proxy (or the defaults for a class of proxies.)

Operationally, each of the stubs and skeletons have been constructed to obtain the injector sequence for each method and to invoke the first injector in that sequence with (1) a (classical CORBA) request object that embodies (a) the target server, (b) the operation to be performed on that server, (c) the arguments of that operation, and (d) a set of annotations for this operation, and (2) the *continuation:* the set of injectors to be executed after this injector. Annotations are name-value pairs, where the name is a string and the value, any CORBA value. (It is the responsibility of annotation users to correctly maneuver in the CORBA type space. For security reasons, future versions of the system

will restrict the annotations, program arguments and results accessed and modified by an injector to those declared for that injector.)

Grossly, an injector wants to perform some actions *before* the server action and some *after*. It is the responsibility of an injector to invoke the remaining injectors of the continuation between its before and after actions (that is, to call the "next" operation on the continuation.) This structure allow injectors to alter the flow of control in interesting ways—for example, to forgo calling the after injectors (as is done in the caching injector, below) and to use the natural exception-catching mechanisms to catch (and correct) exceptions in the continuation processing.

Elsewhere we have argued that the injector mechanism can be used to imbue applications with ilities such as security, reliability, quality of service, and manageability [3]. To make the discussion more concrete, in this section we consider specific injectors we have implemented (or are in the process of implementing) and their impact on these ilities.

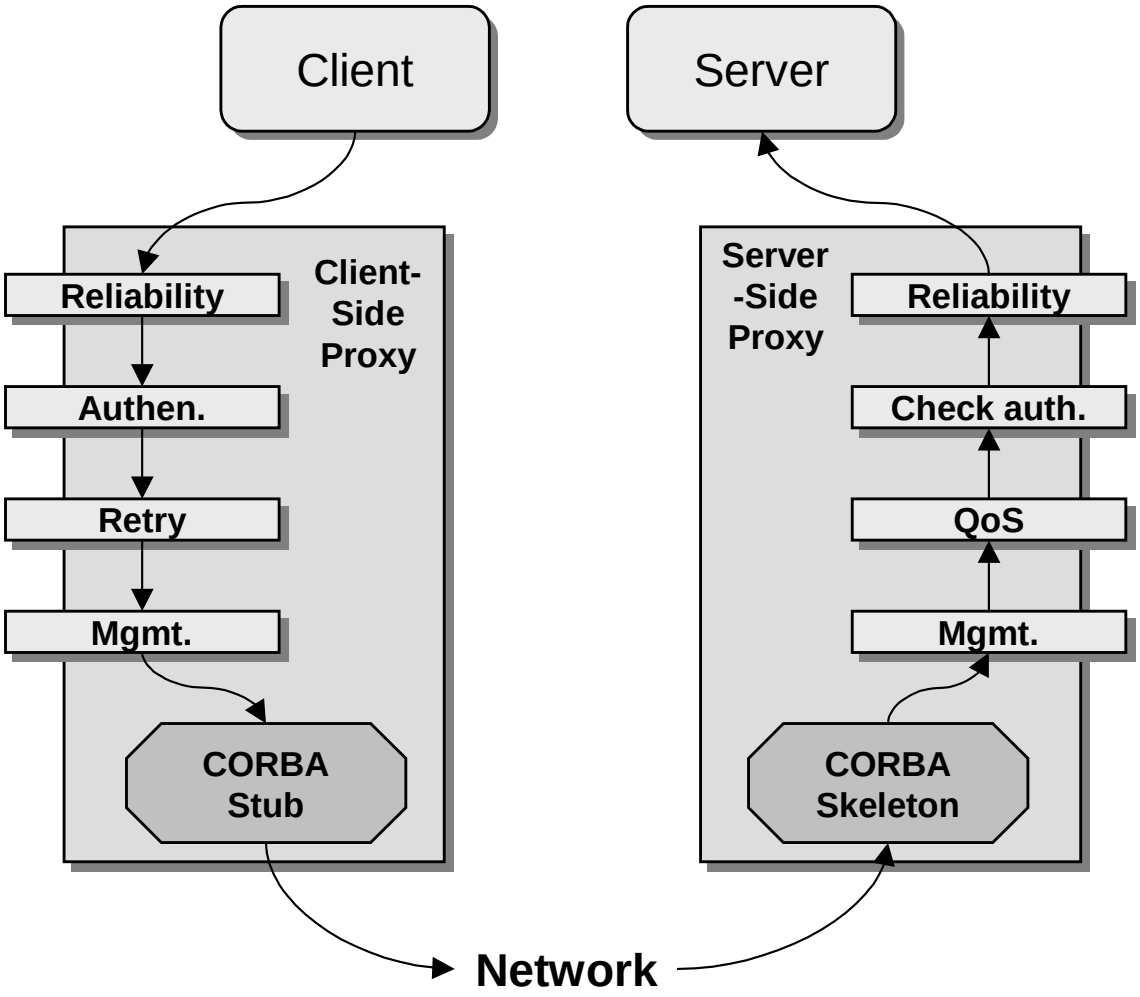


Figure 2: Injectors on stubs and skeletons

### *Security*

Classically, security has four components: (1) *authentication*, determining the identity of a user; (2) *access control*, deciding if a user is permitted to do a specific operation; (3) *encryption*, encoding messages between correspondents; and (4) *intrusion detection*, recognizing attacks on the system. We have developed injectors to do authentication and access control.

The *authentication injector* keeps the user's credentials in a request annotation. The injector runs on both the client and server. If the server-side injector is unhappy with the credentials offered in the annotation, it raises an exception that is caught by the client side. The client injector can then query the user (in our example, demand the user present their Java ring) and restart the request with the newly obtained credential values. (A more thorough version of this injector would encrypt the credential values.)

The *access control injector* simply raises an exception if the user identification annotation does not have the appropriate privileges to perform the specified operation. A more complex version of this injector would be able to examine the arguments of the request and the resulting value, thereby differentiating access with respect to the actual values. For example, a defense-contractor's phone directory service might censor (or even fake) requests for information about people whose employment is meant to be secret, except from trusted sources.

While it is possible to do encryption with injectors, injectors are not the best mechanism. The difficulty lies in encrypting and decrypting references to objects—such things must be recognized as objects by the demarshaling code, and encrypting them after demarshaling would preclude this. We discuss this issue with respect to CORBA interceptors, below.

An intrusion detection injector could be used report suspicious service requests to a security agency [4], though is not in itself an intrusion detection algorithm. Similarly, such injectors could be notified by security agencies of new patterns of suspicious activities to look for, and could be used to reject intrusive requests.

### *Reliability*

We have created two injectors focussed on increased reliability. The *error retry injector* catches errors and retries them (up to some predefined limit). This is useful in situations where the system load or connection noise tends to time out some requests that would otherwise work. By retargeting to a peer of the original service target, the error retry injector could be used to produce redundancy or load balancing. Of course, this strategy is appropriate only on services without harmful side effects—it the error happens after the service, such an injector could be invoking the service repeatedly.

The *rebind injector* notices broken connections and opens a connection to an alternative server. This replicates a provision found in some CORBA ORBs at the system level.

We have also experimented with using replication to maintain a more reliable database. Replication mechanisms could be built into injectors, though it seems better to treat the replication injector as seeking to rebind until replication confirmation is received.

### *Quality of service*

By quality of service we mean to encompass a variety of requirements for getting things done quickly and within time constraints. The real-time community recognizes two varieties of real-time systems, *hard real-time* and *soft real-time*. Hard real-time systems have tasks that must be completed at particular deadlines, or else the system is incorrect. Soft real-time systems seek to allocate resources so as to accomplish the most important things. To achieve hard real-time systems, one can either reserve resources and plan consumption or use some kind of anytime algorithm. Aside from that latter, somewhat esoteric choice, hard real-time requires cooperation throughout the processing chain (for example, in the underlying network), for the promise of particular service can be abrogated in too many places. That is, you can't get hard real-time unless you build your entire system with that in mind. For such systems, we have built a *service reservation injector*, that invokes the appropriate calls to reserve service paths on an ATM network.

Soft real-time quality of service is amenable to several injector mechanisms. We have built a *queue-manager injector* that queues requests to a server and performs higher-priority requests first. Priorities are conveyed as annotations. Since the default behavior is to propagate the annotations of a service to its calls, this mechanism performs chained calls at the priority of the original.

We have also developed a *side-door injector* that open socket connections between the client and server sides for the transport of massive data sets without the overhead of marshaling and the CORBA stack.

Quality of service can also be improved just by making things faster. We have injectors for two methods of overall program performance improvement. The *futures injector* implements futures. Most models of concurrent computing are either synchronous or asynchronous. In a synchronous system the maker of a request waits for the recipient of the request to finish (and perhaps to return an answer). This is the model of telephone calls and procedure calls. Synchronous communication has the advantage that the requestor has an easy time knowing what a response is about (it's right there in the code—this exact spot) but the disadvantage that the requestor has to wait for the answer (when it could be doing other things.) In an asynchronous communication, a requestor sends off a request and then continues about its business. This is like a letter or event-based messaging system. Asynchronous communication has the advantage that the requestor can continue processing while the request is being handled, but the disadvantage that there's no simple way to match responses into the requestor's code.

Futures are a mechanism for bridging the conceptual distance between these two. With futures, requests immediately return a *future*, an object in the requestor's space. Think of this future as a "box." The system will arrange for the reply to the message to go into the box. At any time, the requestor can open the box (try to use the reply). If the reply is there, the requestor can use it (and doesn't have to wait.) If the reply isn't there yet, the requestor waits for the answer to appear. (We also allow an interface for the program to "shake the box" to see if there's anything there yet, but such a program knows it's getting a future—otherwise, the behavior is transparent to the application.) The trickiest part of a futures injector is that the application program wants not only to get back this box with the right properties, but also that the box needs to support the right interface (be the right type of thing with the right type of operations.) Using the futures injector thus requires extending the IDL compiler to create such types.

The caching injector is more straightforward. This injector caches the values of remote calls. When a call to the same underlying object, on the same method, with the same arguments is used again, it retrieves the value from the cache and dispenses with the remote call. Of course, this is appropriate only for services that are functional and free of side-effects, and has the potential for impairing other injectors, such as an accounting injector that wants to measure service use. (As have such caches on the World Wide Web caused trouble for those who wish to count eyeballs on advertisements.)

### *Manageability*

We take a network control perspective on manageability, dividing manageability into five elements: performance measurement, accounting, failure analysis, intrusion detection, and configuration management. We have developed an accounting injector that reports on service use (and totals charges) and a logging injector that can be used to report arbitrary events to a debugging system and to perform updates on graphical user interfaces. We have also developed a smart "publish-and-subscribe" event notification system and incorporated it into the framework to allow the appropriate spread of information about events throughout the system.

We have also begun work on a configuration management injector that dynamically tests for incompatible versions of injectors and automatically updates stale configurations to the current version. The dynamic nature of injectors thus provides the potential for uniformly "self-updating" software, a critical element over the evolution of long-lived systems.

## Related work

We have described a mechanism for separately specifying system-wide concerns in a component-based programming system and then weaving the code handling those concerns into a working application. This is the theme of Aspect-Oriented Programming (AOP). OIF is an instance of AOP, and brings to AOP a particularly elegant division of responsibilities. Key work on AOP includes Harrison and Ossher's work on Subject-Oriented Programming [5] which extends OOP to handle different subjective perspectives; the work of Aksit and Tekinerdogan on message filters [6], which, like OIF, reifies communication interceptors; Lieberherr's work on Adaptive programming [7] which proposed writing traversal strategies against partial specifications; and Kiczales and Lopes [8] work on languages for separate specifications of aspects, which effectively performs mixins at the source-code language level. Czarnecki and Eisenecker's book [9] includes a good survey of AOP technology.

The idea of intercepting communications has occurred several times in the history of computer science. Perhaps the earliest examples were in Lisp: the Interlisp advice mechanism and mix-ins of MacLisp. A more modern realization is seen in mediators [10], which recognizes the implicit agent-hood of the communication interception elements. More recently, the CORBA standard has been extended to provide *interceptors*, programmer-defined operations that run in the communication path. From our point of view, this is the right idea, wrongly implemented. CORBA interceptors run after the call's arguments have been marshaled, making them opaque to the interceptor code (though well-prepared for encryption). CORBA interceptors are also considerably more

structurally rigid than the OIF framework's injectors, not being objects to be manipulated at run-time. If one is particularly fond of CORBA interceptors, one can view our work as a methodology for using them.

Thompson *et. al* [11] present an OIF-like use of injector-like plug-ins in a web architecture. Examples of uses of these plug-ins include performance monitoring and collaborative documents.

It is common to tackle ility concerns by providing a framework with specific choices about those concerns. Examples of such include transaction monitors (e.g., Encina, Tuxedo) and distributed frameworks like Enterprise Java Beans and CORBA.

The use of a separate specification language for creating filters parallels the work at BBN on quality of service [12], where an IDL-like Quality Description Language is woven with IDL to affect system performance.

## Discussion

A problem with large engineering software applications is that the majority of the code is devoted not to implementing desired input-output behavior but to providing system-wide properties like reliability, availability, responsiveness, performance, security, and manageability. While services to support system requirements may be modularized and packaged, the calls to these services must be sprinkled and coordinated throughout the components of the system. This produces code where the original design has been obscured by code to support system requirements. This further requires changing the actual source code in response to system-wide policy changes.

NASA has undertaken several activities to address the issues associated with large-scale distributed applications, particularly supercomputing applications. The NASA Information Technology Base Program [13], and NASA High Performance Computing and Communication Program [14] are coordinating funding of implementations of an "Information PowerGrid." The distributed supercomputing grids aim to build on system-wide systems like Globus [15]. While this looks to be a worthy effort to improve massive computational codes with C and FORTRAN interfaces, the more likely benefactors of distributed, collaborative design usually don't require the same programming interfaces or desire access to be on a supercomputing grid. The programming methods provided by the grid packages instill and promote the same mixing of application and support system requirements, just in an improved fashion.

NASA Ames Research Center's Object Infrastructure Framework (OIF) is an example of Aspect-Oriented-Programming whose goal is to simplify the development of object-oriented, distributed engineering software systems by providing an architecture that separates code that supports input-output behavior from code that achieves system requirements. This then allows for independent development of these two different concerns. The CORBA/Java implementation is a toolkit that addresses the popular and effective programming object/language paring.

As distributed large-scale analysis, design and engineering programs are developed for NASA missions in 21$^{st}$ century, more intelligent software frameworks will be used to rapidly provide the high degree of robust applications NASA's Missions and Programs needs. The OIF is such a framework and will be used in several NASA distributed applications supporting the Intelligent Synthesis Environment, and Information Technology Base Programs. We are currently applying the OIF framework to the

implementation of the third version of the DARWIN system [16], a tool for the management, distribution, and analysis of aerospace (wind-tunnel, and CFD) data available to researchers. This version of DARWIN will rely on OIF to improve robustness, security, managability and overall performance.

In closing, we note that annotation is a key concept for the ISE environment. The work described here is concerned with annotating communications between active processing elements, but that is not the limit of the uses of annotation. An ISE will contains many possible data sets, with varying dependencies, configurations details, ownership rights, and so forth. Keeping track of such annotations, responding to the annotations in processing, and asserting the correct annotations for derived objects will prove an important enabling principle of ISE-like environments.

## Acknowledgments

The ideas expressed in this paper have emerged from the work of the MCC Object Infrastructure Project. We thank Stu Barrett, Carol Burt, Deborah Cobb, T.W. Cook, Phillip Foster, Andre Goforth, Barry Leiner, Ted Linden, Erik Mettala, David Milgram, Steve Rodgers, Gabor Seymour, Tom Shields, Doug Stuart and Craig Thompson for their contributions.

## References

1. Goldin D. S. Venneri S. L. Noor A. K. Beyond Incremental Change. IEEE Computer 1998; 31(10): 31-39.
2. Siegel, J. CORBA: Fundamentals and Programming, New York: Wiley, 1996.
3. Filman R. Barrett S. Lee D. Linden T. Inserting Ilities by Controlling Communications. Submitted. http://www.mcc.com/projects/oip/papers/ins_ilities.htm
4. Filman R. Linden T. Communicating Security Agents. The Fifth IEEE-Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises—International Workshop on Enterprise Security. Stanford, California, June 1996, 86-91.
5. Harrison W. Ossher H. Subject-Oriented Programming (A Critique of Pure Objects). Proc. OOPSLA ’93. ACM SIGPLAN Notices 1993; 28 (10): 411-428.
6. Aksit M. Tekinerdogan B. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. AOP '98 workshop position paper, 1998. http://wwwtrese.cs.utwente.nl/Docs/Tresepapers/FilterAspects.html
7. Lieberherr K. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, Boston: PWS Publishing Company, 1996.
8. Kiczales G. Lamping J. Mendhekar A. Maeda C. Lopes C. Loingtier J.-M. Irwin J. Aspect-Oriented Programming. Xerox PARC Technical Report, February 97, SPL97-008 P9710042. http://www.parc.xerox.com/spl/projects/aop/tr-aop.htm
9. Czarnecki. K. Eisenecker, U. Generative Programming: Methods, Techniques, and Applications, Reading, Massachusetts: Addison-Wesley, 1999.
10. Wiederhold G. Mediators in the Architecture of Future Information Systems. IEEE Computer 1992; 25 (1): 38-49.
11. Thompson C. Pazandak P. Vasudevan V. Manola F. Palmer M. Hansen G. Bannon T. Intermediary Architecture: Interposing Middleware Object Services between Web

Client and Server" To appear in Computing Surveys. http://www.objs.com/OSA/Intermediary-Architecture-Computing-Surveys.html
12. Schantz R. Bakken D. Karr D. Loyall J. Zinky J. Distributed Objects with Quality of Service: An Organizing Architecture for Integrated System Properties. OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, California, Jan. 1998. http://www.objs.com/workshops/ws9801/papers/paper099.doc
13. NASA Information Technology Base Program, http://science.nas.nasa.gov/IT/
14. High Performance Computing and Communications Program, http://science.nas.nasa.gov/HPCC/
15. Globus, http://www.globus.org/
16. Joan Walton, Robert E. Filman, and David J. Korsmeyer. "The Evolution of the DARWIN System." to appear in 2000 ACM Symposium on Applied Computing March 2000, Como, Italy. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/darwin/evolution-of-darwin.pdf

## Vitae

**Robert E. Filman** is a Computer Scientist with Caelum Research Corporation on the staff of NASA Ames Research Center. Prior to joining NASA, he was on the staff of Lockheed Martin, IntelliCorp and Hewlett Packard, and on the faculty of Indiana University, Bloomington. Dr. Filman is the author (with Daniel P. Friedman) of *Coordinated Computing: Tools and Techniques for Distributed Software* (McGraw-Hill), is the Associate Editor-in-Chief of IEEE Internet Computing, and has published in the areas of software engineering, distributed computing, network security, programming languages, artificial intelligence, and human-machine interface. He received his B.S. (Mathematics), M.S. (Computer Science) and Ph.D. (Computer Science) from Stanford University.

**David J. Korsmeyer** is a Senior Project Scientist in the Computational Sciences Division at NASA Ames Research Center. He is the Level 2 Manager within NASA's Information Technology Base Program for the aptly named Analytical Tools and Environments for Design Element and lead the development of DARWIN. Dr. Korsmeyer was recently chosen as the Chief IT Architect for Science and Engineering Infrastructure by NASA's Chief Information Officer (CIO) and Chief Technologist Office (CTO). Dr. Korsmeyer is also the Special Assistant for Information Technology for the ISE Program Office. He received his B.S. (Aerospace Engineering) from the Pennsylvania State University, and his M.S. (Aerospace Engineering) and Ph.D. (Aerospace Engineering) from the University of Texas at Austin.

**Diana D. Lee** is a Computer Scientist with Caelum Research Corporation on the staff of NASA Ames Research Center. Prior to joining NASA, she was on the staff of Microelectronics and Computer Technology Corporation, Lockheed Martin, and International Business Machines. Ms. Lee received her B.A. (Mathematical Sciences) from Rice University and her M.S. (Applied Mathematics) from Santa Clara University. Ms. Lee is also an adjunct faculty member in the Graduate School of Engineering at Santa Clara University.